Introduction to coding

HTML Version: https://stirlingcodingclub.github.io/getting_started/notes.html

Brad Duthie

10 OCT 2019

The purpose of this introduction

The purpose here is to get readers past the initial learning curve of coding as quickly as possible. If you want to start coding for yourself, particularly in R for data analysis, but are not sure where to start, then read on. By the end of these notes, you should be able to navigate through the basic graphical user interface of Rstudio and write some basic lines of code. The goal is not to develop proficiency in coding or R yet, but to help you get to the point at which it is possible to write and run code, make coding mistakes, and learn from other researcher's code.

Why use the R programming language?

The computer programming language R is a powerful and very widely-used tool among biologists for analysing data. You can use it to analyse and plot data, run computer simulations, or even write slides, papers, or books. The R programming language is completely free and open source, as is the popular Rstudio software for using it. It specialises in statistical computing, which is part of the reason for its popularity among scientists.

Another reason for its popularity is its versatility, and the ease with which new techniques can be shared. Imagine that you develop a new method for analysing data. If you want other researchers to be able to use your method in their research, then you could write your own software from scratch for them to install and use. But doing this would be very time consuming, and a lot of that time would likely be spent writing the graphical user interface and making sure that your program worked across platforms (e.g., on Windows and Mac). Worse, once written, there would be no easy way to make your program work with other statistical software should you need to integrate different analyses or visualisation tools (e.g., plotting data). To avoid all of this, you could instead just present your new method for data analysis and let other researchers write their own code for implementing it. But not all researchers will have the time or expertise to do this.

Instead, R allows researchers to write new tools for data analysis using simple coding scripts. These scripts are organised into R packages, which can be uploaded by authors to the Comprehensive R Archive Network (CRAN), then downloaded by users with a single command in R. This way, there is no need for completely different software to be used for different analyses – all analyses can be written and run in R.

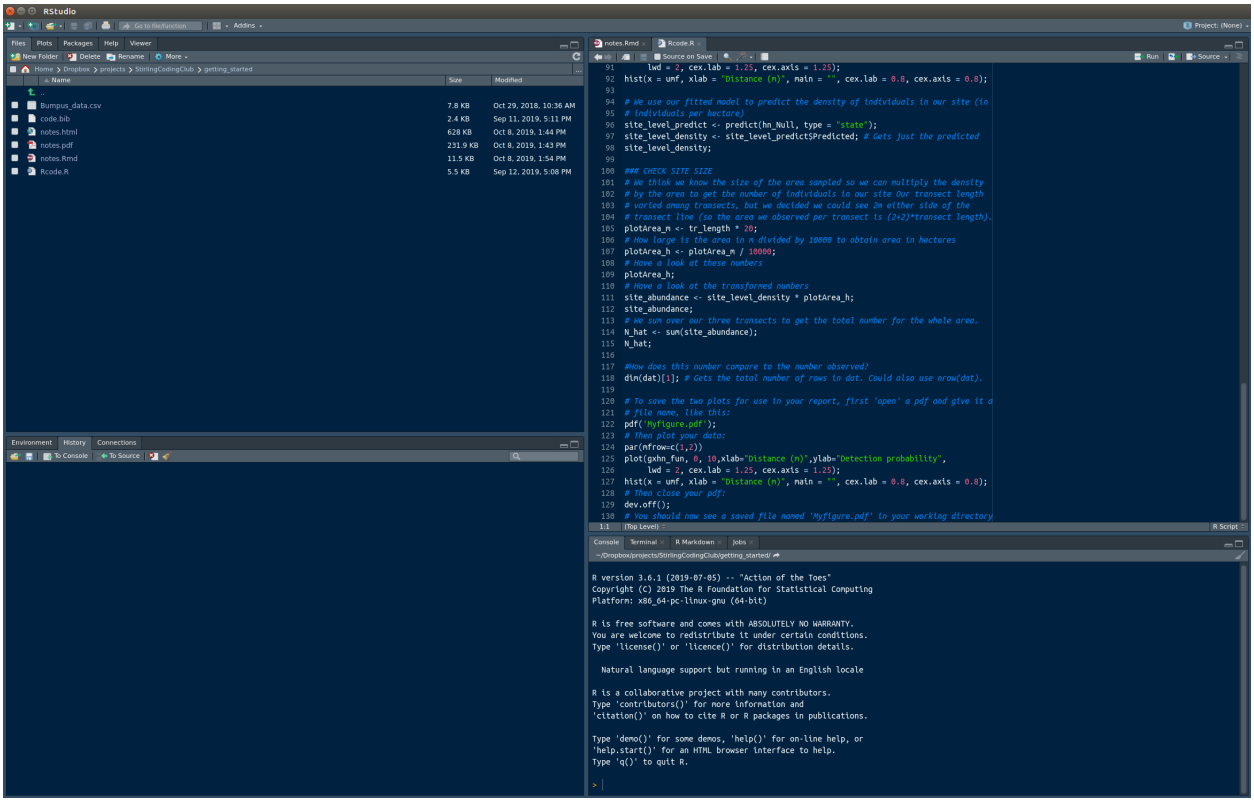
The downside to all of this is that learning R can be a bit daunting at first. Running analyses is not done by pointing and clicking on icons as in Excel, SigmaPlot, or JMP. You need to use code. Here we will start with the very basics and work our way up to some simple data analyses.

Getting started in R, and the basics

Installation. The first thing to do is download Rstudio if you have not already. Note that R and Rstudio are not the same thing; R is a language for scientific computing, and can be used outside of Rstudio. Rstudio is a very useful tool for coding in the R language. As a loose analogy, R is like a written language (e.g., English, Spanish) that can be used to write inside Rstudio (e.g., a word processor such as Microsoft Word, LibreOffice). But by downloading Rstudio, you will also download the R programming language. Look

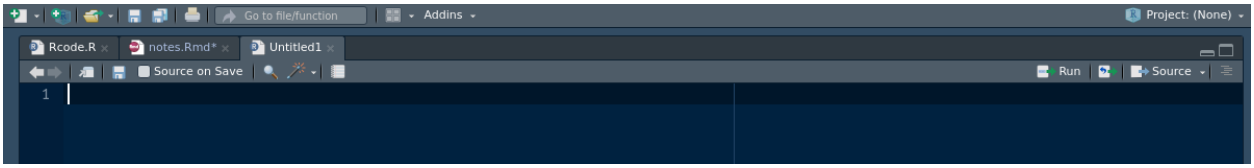
carefully at the version of Rstudio that you download; different installers exist for Windows, Mac, and Linux. The most recent version of Rstudio requires a 64-bit operating system. Unless your computer is quite old (over seven years), you most likely have a 64-bit operating system rather than a 32-bit operating system, but if you are uncertain, then it is best to check.

Running Rstudio. When you first run Rstudio, you will see several windows open. It will look something like the below, except probably with a standard black on white theme (if you want, you can change this by selecting from the toolbar 'Tools > Global Options...', then selecting the 'Appearance' tab on the left).



This might look a bit intimidating at first. Unlike Microsoft Excel, SigmaPlot, or JMP, there is no spreadsheet that opens up for you. You interact with R mostly by typing lines of commands rather than using a mouse to point and click on different options. Eventually, this is liberating, but at first it will probably feel overwhelming. First, let us look at all of the four panes in the Figure above. Your panes might be organised a bit differently, but the important ones to start out with are the 'Source' and the 'Console'. These are shown in the right hand panes in the above Figure.

To make sure that the Source pane is available to you, open an new R script by selecting from the toolbar 'File > New File > Rscript' (shortcut: Shift+Ctrl+N). You should see a new Rscript open up that looks something like the below (the colour scheme might differ).



Think of this Source file like a Word document that you have just opened up – completely blank and ready for typing new lines of command to read in data and run analyses. We will come back to this Source file, but for now just know that the Source file stores commands that we want R to intepret and use. The Source file does this by sending commands to the R console, which we will look at now.

The R console should be located somewhere in Rstudio (I like to keep it directly underneath my R Source files). You can identify it by finding the standard R information printed off, which should look something like the below.

```
R version 3.6.1 (2019-07-05) -- "Action of the Toes"
Copyright (C) 2019 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

The console is where all of the code is run. To get started, you can actually ignore everything else and just focus on this pane. If you click to the right of the greater than sign `>`, you can start using R right in the console. To get a feel for running code in the console, you can use the R console as a standard calculator. Try typing something like the below to the right of the `>`, then hit 'Return' on your keyboard (note, all of my semi-colons are optional).

```
2 + 5;
```

```
## [1] 7
```

Now try some other common mathematical operations, one line at a time.

```
4 * 4;
```

```
## [1] 16
```

```
12 - 3;
```

```
## [1] 9
```

```
5^2;
```

```
## [1] 25
```

Notice that R does the calculation of each of the above mathematical operations and returns the correct value on the line below. If you are familiar with using Microsoft Excel, this is the equivalent to typing `= 2 + 5`, `= 4 * 4`, etc., into a cell of an Excel spreadsheet. You might also be familiar with spreadsheet functions as well, such as the square root function, which you could use in Excel by typing, e.g., `= sqrt(25)` into a spreadsheet cell. This works in the R console too; the functions actually have the same syntax, so you could type the below into the console and hit 'Enter'.

```
sqrt(256);
```

```
## [1] 16
```

The console returns the correct answer 16. Similar functions exist for logarithms (`log`) and trigonmetric functions (e.g., `sin`, `cos`), as they do in Microsoft Excel. But this is just the beginning in R. Functions can

be used to do any number of tasks. Some of these functions are built into the base R language, others are written by researchers and distributed in R packages, but you can also learn to write your own R functions to do any number of customised tasks. You will need to use functions in nearly every line of code you write (technically, even the `+`, `-`, etc., are also functions), so it is good to know the basics of how to use them.

Most functions are called with open and closed parentheses, as in the `sqrt(256)` above. The `sqrt` is the function, while the `256` is a function argument. An argument is a specific input to a function, and functions can take any number of arguments. For the `sqrt` function, only one argument is needed, but many arguments will have more than one argument. For example, if we want to take the logarithm of some number using the `log` function, we might need to specify the base. In this case, we clarify the number for which we want to compute the log `x` and the logarithm base `base`. Let us say that we want to compute the logarithm of 256 in base 10.

```
log(x = 256, base = 10);
```

```
## [1] 2.40824
```

Note that some arguments are required, while some are optional. In the case of `log`, the first argument is required, but the base is actually optional. If we do not specify a `base`, then the function simply defaults to calculating the natural logarithm (i.e., base e). Hence, the below also works (note that we get a different answer because the bases differ).

```
log(x = 256);
```

```
## [1] 5.545177
```

In fact, we do not even need to specify the `x` because only one argument is needed for the `log` function. Hence, if only one argument is specified, the function just assumes that this argument is `x`. Try the below.

```
log(256);
```

```
## [1] 5.545177
```

Note that functions can be nested inside other functions, though this can get messy. For example, if you wanted to get the logarithm of the logarithm of 256, then you could write the below.

```
log( log(256) );
```

```
## [1] 1.712929
```

Also note that functions do not need to be mathematical in R; they do not even need to operate on numbers. One very useful function is the `help` function, which provides documentation for other R functions. If, for example, we were not sure what `log` did, or what arguments it accepted, then we could run the code below.

```
help(topic = log);
```

Try running the above line of code in the R console. You should see a description of the `log` function, along with some examples of how it is used and the two arguments (`x` and `base`) that it accepts. Anytime you get stuck with a function, you should be able to use the `help` function for clarification. You can even use a shortcut that returns the same as the `help(topic = log);` above.

```
?log;
```

We now have looked at three functions, `sqrt`, `log`, and `help`. If you have previous experience with Microsoft Excel spreadsheets, you should now be able to make the conceptual connection between typing `=sqrt(25)` into a spreadsheet cell and `sqrt(25)` into the R console. You should also recognise that R functions serve a much broader set of purposes in R. Next, we will move onto assigning variables in the R console.

Assigning variables in the R console

In R, we can also assign values to variables using the characters `<-` to make an arrow. Say, for example, that we wanted to make `var_1` equal 10.

```
var_1 <- 10;
```

We can now use `var_1` in the console.

```
var_1 * 5; # Multiplying the variable by 5
```

```
## [1] 50
```

Note that the correct value of 50 is returned because `var_1` equals 10. Also note the comment left after the `#` key. In R, anything that comes after `#` on a line is a comment that R ignores. Comments are ways of explaining in plain words what the code is doing, or drawing attention to important notes about the code.

Note that we can assign multiple things to a single variable. Here is a vector of numbers created using the `c` function, which combines multiple arguments into a vector or list. Below, we combine six numbers to form a vector called `vector_1`.

```
vector_1 <- c(5, 1, 3, 5, 7, 11); # Six numbers
```

We can now print and perform operations on `vector_1`.

```
vector_1;      # Prints out the vector
```

```
## [1]  5  1  3  5  7 11
```

```
vector_1 + 10; # Prints the vector plus ten
```

```
## [1] 15 11 13 15 17 21
```

```
vector_1 * 2;  # Prints the vector times two
```

```
## [1] 10  2  6 10 14 22
```

```
vector_1 <- c(vector_1, 31, 100); # Append the vector
vector_1;
```

```
## [1]   5   1   3   5   7  11  31 100
```

We can also assign lists, matrices, or other types of objects using the `list` function.

```
object_1 <- list(vector_1, 54, "string of words");
object_1;
```

```
## [[1]]
## [1]   5   1   3   5   7  11  31 100
##
## [[2]]
## [1] 54
##
## [[3]]
## [1] "string of words"
```

Play around a bit with R before moving on, and try to get confortable using the console. When you have finished with the R console, continue reading to learn how to store lines of code using an R script.

Reading in data

Now we need to read in the data collected from out in the field. Make sure that the file is in the same place as your current working directory (`getwd()`). I have named my file 'MyTransect.csv', but yours might be different.

```
dat <- read.csv(file = "Bumpus_data.csv", header = TRUE);
```

Note above that I have used the 'read.csv' function to read the CSV file into the variable `dat` in R. To make sure that everything looks correct, you can type `dat` in the console to see all of the data print out. I will use the 'head' function below to just print off the first six rows.

```
head(dat);
```

```
##    sex  surv totlen wingext  wgt head humer femur tibio skull stern
## 1 male alive    154     241 24.5 31.2 0.687 0.668 1.022 0.587 0.830
## 2 male alive    160     252 26.9 30.8 0.736 0.709 1.180 0.602 0.841
## 3 male alive    155     243 26.9 30.6 0.733 0.704 1.151 0.602 0.846
## 4 male alive    154     245 24.3 31.7 0.741 0.688 1.146 0.584 0.839
## 5 male alive    156     247 24.1 31.5 0.715 0.706 1.129 0.575 0.821
## 6 male alive    161     253 26.5 31.8 0.780 0.743 1.144 0.607 0.893
```

If the data appear to be read into R correctly, then you can move on to the analysis below. The most common error at this point is trying to read a CSV file into R that does not exist, either because the file is misspelled or the directory is incorrect. **Note that everything in R is case sensitive**, meaning that if a letter is capitalised in a file name or a variable, then it needs to be capitalised when you write it out.

Now that we have the data read in and assigned to the variable `dat`, we can summarise it in some different ways. Try typing the functions below into the R console to see what happens. **Note that your data will be different than what is in this document. This means that your plots and analyses should look different, even though the code will be the same.**

```
names(dat);       # Get the column names in dat
str(dat);         # Compactly display the structure of dat
attributes(dat);  # Get the attributes of dat (e.g., col and row names)
dim(dat);         # How many rows and columns in dat?
summary(dat);     # Summary statistics of dat (e.g., min, max, mean)
```

Note that if you are not sure what a function does, you can use the 'help' function in the console to find out. The 'help' function will bring up the function's documentation. For example, say you were not sure what the function 'attributes' does, or what arguments it takes.

```
help(attributes); # See 'attributes' documentation: ?attributes also works
```

We can also summarise the data visually using the generic 'plot' function. R will recognise the structure of the data and create a box and whisker plot that summarises the distances for each of our three transects.

We can also make a histogram of the distances that we collected. This is a good way to check for outliers. If you end up with a distance that is much too far to be realistic, then check to make sure that you did not type it into Excel incorrectly.

Appendix: R script file

```
# Here is your first R script.
```

References